View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universiteit Twente Repository

1

USING A FUNCTIONAL NOTATION TO SPECIFY ABSTRACT SIMULATION MODELS

H.L. Muller* **P.H. Hartel** **L.O. Hertzberger**

University of Amsterdam,
Kruislaan 403,
1098 SJ Amsterdam,
Holland

Abstract

Simulation is the process where a real world entity is modelled and implemented on a computer. It turns out that all simulations in all application domains use only a few different simulation algorithms. In this article four of these algorithms are derived informally. Starting with a definition of a simulation specified in a functional language, a demand driven simulation algorithm, continuous time, discrete time and discrete event algorithms are subsequently derived. The advantage of using a functional language to specify these algorithms is that it is relatively simple to reason about several aspects, as correctness, efficiency, expressiveness, and potential parallelism.

The algorithms are not functionally equivalent, they have increasing expressiveness and increasing efficiency at the cost of decreasing potential parallelism. This is illustrated with the help of a trivial example from the field of logic circuit design, although the algorithms can be used for other (large) applications without modification. The four simulation algorithms have been implemented and executed to verify the efficiency considerations in practice.

1.1 Introduction

Many simulation tools have been developed over the past decades. Some tools are specifically written for one application (the simulation of a certain type of microprocessor), other tools are useful for a specific application domain, such as fluid dynamics, digital signal processing architectures or queueing problems. Despite the rich variety of simulators, all are based on the same simulation principles, which will be discussed in detail. Most simulators have one problem: the vast amount of CPU time needed. To provide insight in the fundamental problems involved in developing efficient simulation methods, a formal, functional description of each of the basic

*Now with: Computer science department, University of Bristol, 10 Priory Road, Bristol BS8 1TU, England. E-mail: muller@acrc.bristol.ac.uk

simulation algorithms is given. These descriptions are analysed with respect to the expressiveness, efficiency, and potential parallelism. Because a formal description is used to specify the algorithms, it is possible to reason about these properties(Kelly 1989, Sijtsma 1989).

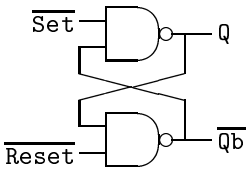
Reasoning about a description and its properties is facilitated by viewing the description as an abstract machine. Execution on an abstract simulation machine corresponds to executing a simulation. The abstract machine view of simulation algorithms allows a clear separation to be made between the specification of the simulation systems and the specification of the simulated object. The first plays the role of the abstract machine, the latter plays the role of an abstract machine program, as is explained in Section 1.2.

Section 1.3 presents the essence of simulation. The resulting algorithm will never be used for real simulations because of grave inefficiencies. From this inefficient algorithm, a simulation algorithm can be derived with a continuous time scale, as presented in Section 1.4. Continuous time simulators are mostly used to model physical processes, where variables change continuously in time. The algorithm can also be turned into a discrete time simulator (Section 1.5), which is more efficient when processes with discrete steps are to be simulated, as for example logic circuits, or problems with queues. In Section 1.6 the event driven simulator is presented, which is the most efficient simulation algorithm for discrete time problems. The paper is concluded with a discussion of the most important properties of the four algorithms.

1.1.1 *Running example and notation*

Throughout the paper, the *Flip-Flop* is used as a running example. The Flip-Flop is one of the elementary circuits that can be used for data storage. The schematics of the Flip-Flop are depicted in Fig. 1.1, together with the truth table. Although a Flip-Flop is not a complex circuit, it possesses all hard problems for simulation: it contains a loop, it contains state, and it will not stabilise unless used in the right way. The Flip-Flop has two inputs labelled $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$, and two outputs, `Q` and $\overline{\texttt{Qb}}$. For a stable situation both $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$ are kept high, giving a high signal on one of the outputs and a low signal on the other: $\overline{\texttt{Set}} = \overline{\texttt{Reset}} = 1 \Rightarrow \texttt{Q} = \neg\overline{\texttt{Qb}}$. Driving $\overline{\texttt{Set}}$ low for a while causes `Q` to become high regardless of its previous state. Driving $\overline{\texttt{Reset}}$ low for a while causes `Q` to become low (and $\overline{\texttt{Qb}}$ high) regardless the previous state. Driving $\overline{\texttt{Set}}$ or $\overline{\texttt{Reset}}$ low for a short period of time brings the Flip-Flop in an oscillating state for an indefinite period of time: it causes a short pulse to start racing through the two nand gates.

In Miranda(TM of Research software Ltd.) the high and low signals are represented by `H` and `L` respectively, while the `X` stands for an undefined signal that can be either high or low. All program fragments use the following definition for the type representing the state of a wire called `threestate`



$\overline{\texttt{Set}}$	$\overline{\texttt{Reset}}$	Q	$\overline{\texttt{Qb}}$
1	1	Q	$\overline{\texttt{Qb}}$
0	1	1	0
1	0	0	1
0	0	1	1

FIG. 1.1. The Flip-Flop circuit and truth table.

and the definition of the nand characteristics (refer to (Bird and Wadler 1988) for an introduction to functional programming, and to (Turner 1985, Turner 1990) for the definition of Miranda):

```
> timestamp == num           || The time is stored as an integer
> threestate ::= X | L | H   || undefined, low, high
>
> nandfun :: threestate -> threestate -> threestate
>
> nandfun H H = L            || The only way to get 'Low' out of a nand
> nandfun L b = H            || 'Low' on left port forces a high output
> nandfun a L = H            || 'Low' on right port forces a high output
> nandfun a b = X            || all other inputs result in undefined
```

The function `nandfun` takes two values of the type `threestate` and produces a value of the same type. When applied to two High input values (`nandfun H H`), the result is a low output (`L`), a low value on the first or the second parameter results in a high output, while all other inputs (for example `nandfun H X`) result in undefined, `X` (note that `a` and `b` are free variables).

The Flip-Flop is simulated, under the assumption that the nand gates introduce a fixed delay of 3 time units. The $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$ wires are driven by the two clock signals that are depicted in Fig. 1.2, together with the expected output signals on `Q` and $\overline{\texttt{Qb}}$. The signals on the `Q` and $\overline{\texttt{Qb}}$ wires are initially undefined, which is shown by the shaded areas. The first change is a transition of the $\overline{\texttt{Set}}$ wire from low (0) to high (1) at time $t = 7$. This has no further effect on the state of the circuit. The second change at time

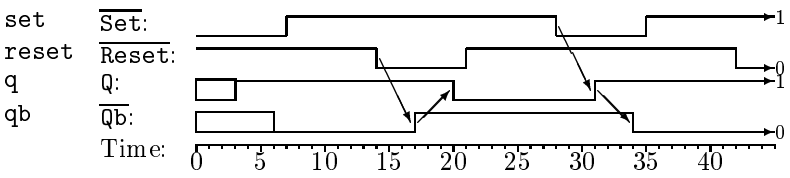


FIG. 1.2. The inputs and outputs of the examples, the shaded parts denote an undefined signal.

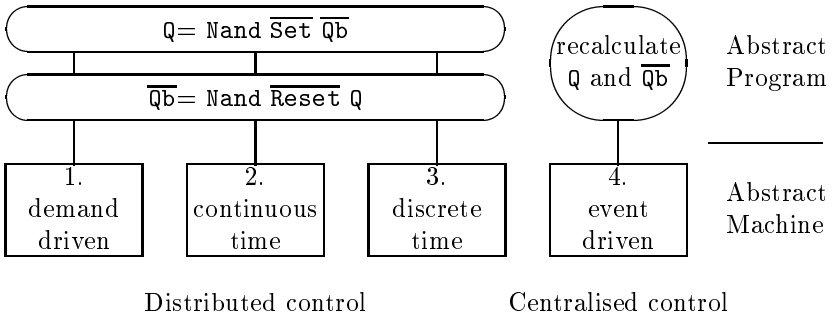


FIG. 1.3. The simulations as two different abstract programs on four different abstract machines

$t = 14$ does have an effect on the state of the circuit, which is shown by the two arrows, that represent the causality of the sequence of events. The transition on the $\overline{\texttt{Reset}}$ wire first causes a transition on the $\overline{\texttt{Qb}}$ wire with a delay of 3 time units. This in turn causes a transition on the Q wire with another delay of 3 units. The system is simulated through another transition of the $\overline{\texttt{Set}}$ wire and another transition on the $\overline{\texttt{Reset}}$ wire.

1.2 An abstract machine program for the Flip-Flop

A system of interacting components, such as the Flip-Flop, can be described by using recursion equations. Assuming that the signals $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$ are defined externally, two equations are necessary to fully define the Flip-Flop:

$$\begin{aligned} \texttt{Q} &= \texttt{Nand}\ \overline{\texttt{Set}}\ \ \overline{\texttt{Qb}} \\ \overline{\texttt{Qb}} &= \texttt{Nand}\ \overline{\texttt{Reset}}\ \texttt{Q} \\ \overline{\texttt{Set}} &= \ldots \\ \overline{\texttt{Reset}} &= \ldots \end{aligned}$$

The true interpretation of the signal wires is the ensemble of all possible values on the signal wires at every possibly instant in time. Simulation provides an approximation to the values at approximated time instances. Because all values of interest are approximated, simulation on a computer is feasible.

The recursion equations that describe a system, which is to be simulated, can be viewed as an abstract program. Such a program may be fed to a certain number of abstract machines that differ in the way in which simulation is performed. All abstract machines should give correct results. The results may be different only in the level of detail provided or in the time it takes to perform a simulation. Fig. 1.3 shows the four different abstract simulation machines. Each provides an implementation of the Nand abstract machine instruction. This implementation performs

```
> wire == timestamp -> threestate
>
> || ================ ABSTRACT MACHINE PROGRAM ================
> q, qb :: wire                 || Definition of the q and qb
> q       = nand1 set   qb      || Define upper nand
> qb      = nand1 reset q       || Define lower nand
>
> || ============ INPUT OF ABSTRACT MACHINE PROGRAM ============
> set, reset :: wire            || Definition of set and reset
> set   t = L,  if t mod 28 < 7 || This defines a clock
>         = H,  otherwise       || 7 ticks low, 2 high
> reset t = set (t+14)          || reset is shifted set.
>
> || =================== ABSTRACT MACHINE ====================
> delay :: timestamp -> timestamp
> delay t     = t-3
>
> nand1 :: wire -> wire -> timestamp -> threestate
> nand1 x y t = nandfun (x tbefore) (y tbefore), if t > 0
>             = X,                               otherwise
>               where
>               tbefore = delay t
```

FIG. 1.4. The source code in Miranda for a demand driven simulator.

the simulation of the circuit based on the data structures used by the particular abstract machine. Although each abstract machine uses a different data representation and mode of execution, no changes should be required to the abstract program. We shall work through each of the four basic simulation algorithms and see how faithful we can be to this principle.

1.3 Demand driven simulation

The Miranda program of Fig. 1.4 shows a demand driven simulator. The approximations to the real values of `Q`, $\overline{\texttt{Qb}}$, $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$ are represented by the functions `q`, `qb`, `set` and `reset` respectively. The type signatures show that these functions map approximated time (of type `timestamp`) onto approximated values (of type `threestate`). As required, the abstract program (i.e. the recursion equations) are literally present in the Miranda program. The `nand1` function and the associated data structures represent the abstract machine.

The state on the wires at a moment t in time is defined in terms of the output of the nand gate at time t, while the nand-function at time t depends on the state at the input wires at time (*delay* t). When the simulator is asked for the state of `Q` at time 40, all states of `Q` and $\overline{\texttt{Qb}}$ are recursively calculated until the moment the simulator comes to a well

defined state (when one of the input signals is low, the output of the nand is fixed regardless its previous state), or until time zero, when the signal is `X` (by definition). In this example, the calculation of `q 40` requires the values of `set` and `qb` of three steps earlier (the delay of the nand gate), the value of `set 37` (which is `H`) and the value of `qb 37`. To calculate `qb 37`, the value of `reset 34` (`H`) and the value of `q 34` is required. This depends on the value of `set 31`, which is `L`; consequently `q 34` equals `H`, `qb 37` equals `L`, and `q 40` thus equals `H`, which is the answer.

An interesting aspect of the demand driven simulator is that by expressing it in a lazy functional language, the mode of evaluation required by the simulation is the natural mode of evaluation provided by lazy evaluation. The language implementation guarantees that the computations as described above proceed in exactly the way described. Most importantly, no states are calculated that are not strictly required, and all expressions can be evaluated in parallel. In spite of this apparent efficient behaviour, demand driven simulation is not used in practice because of three serious drawbacks. Firstly it outputs only the state at a given moment in time, the question: "What is the first time that `Q` will become high after time T" cannot be answered directly, only an exhaustive search can provide the answer. To print the history of state changes on both `q` and `qb` one would apply each to all time stamps of interest, as is written in Miranda with the `map` function:

```
> map q [0..27]  = [X,X,X,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,L,L,...
> map qb [0..27] = [X,X,X,X,X,X,L,L,L,L,L,L,L,L,L,L,L,H,H,H,H,H,...
```

The second drawback is that the recursive calculation places a huge demand on the memory, since a whole stack of calculations is built before they are evaluated. Thirdly, in a more complex circuit, where `Q` is used in more than one place, `Q` will be recalculated each time leading to an exponential time consumption. All three drawbacks can be relieved by reversing the order of computations, thus by starting with the state at time zero, and by processing forward in time. By memoising (Hughes 1985) the old states in (lazy) lists, states in the past can be referred to. There are two ways to maintain these lists: with implicit timing information, giving a *continuous* time simulation, or with explicit time stamps, resulting in a *discrete* time simulation. These methods are presented in the next sections.

1.4 Continuous time simulation

A simulation of the Flip-Flop with an implicit continuous time increment is an improvement over the demand driven simulator (Vree 1989). The wires are now represented by infinite lists (streams) of states. The i^{th} element of such a list represents the value of the wire at time $i \times \delta_t$, where δ_t is the time increment. The components are modelled by functions working on these lists, as synchronous processes (Kahn 1974). A nand-process in the

```
> || ================ ABSTRACT MACHINE PROGRAM ================
> q, qb :: [threestate]
> q    = nand2 set   qb
> qb   = nand2 reset q
>
> || ============ INPUT OF ABSTRACT MACHINE PROGRAM =============
> set, reset :: [threestate]
> set  = [L,L,L,L,L,L,L,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H]
> reset= [H,H,H,H,H,H,H,H,H,H,H,H,H,H,L,L,L,L,L,L,L,H,H,H,H,H,H,H]
>
> || ==================== ABSTRACT MACHINE =====================
> delay :: [threestate] -> [threestate]
> delay  xs            = X:X:X:xs
>
> nand2, nand2' :: [threestate] -> [threestate] -> [threestate]
> nand2' x      []     =  []                          || terminate
> nand2' []     x      =  []                          || terminate
> nand2' (x:xs) (y:ys) =  nandfun x y : nand2' xs ys || apply nand
> nand2  xs     ys     =  delay (nand2' xs ys)
```

FIG. 1.5. The Flip-Flop in a continuous time simulator.

Flip-Flop thus takes two *lists* of states as input parameters and produces a list of states as output. The source code for a continuous simulation of the Flip-Flop is shown in Fig. 1.5. Comparing this with Fig. 1.4, shows that the abstract program has not been altered, although the abstract machine and the data structures involved are now completely different.

The function `nand2` consumes states from two input streams, and produces the output-stream with help of the function `nandfun` as defined earlier. The function `nand2` starts with three undefined states on the output list (`X:X:X`) to model a delay of three time steps. As before, the four wires are named `q`, `qb`, `set` and `reset`, and are connected by means of two nand gates. By providing two input lists for the $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$, the program computes the approximated output values on the streams `Q` and $\overline{\texttt{Qb}}$:

```
> q  = [X,X,X,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,L,L,L,L,L,L,L,L,L]
> qb = [X,X,X,X,X,X,L,L,L,L,L,L,L,L,L,L,H,H,H,H,H,H,H,H,H,H,H,H,H]
```

Although the lists `q` and `qb` are mutually dependent, the algorithm does not deadlock because the start elements of the lists are defined (`X:X:X`). The productivity theory of (Sijtsma 1989) provides a framework to reason about liveness of functional programs. In terms of this theory, the function `nand2` is +3-productive, indicating that a (cyclic) network of `nand2` functions is productive (which means that the network will keep producing elements as long as input is provided on the input lists). Since the functions operating on the streams are independent, the simulation can be parallelised easily.

An essential property of continuous time simulators is that the time step is constant: all processes are synchronous, the states are produced synchronously, implying that all components of the simulation use the same time step. This type of simulator is used in simulations where the state would change continuously in time, but where the state is necessarily modelled with small discrete steps because the time cannot be incremented continuously. This technique is amongst others applied in (Vree 1989) for the simulation of water heights. Other examples include the simulation of electrical currents in a circuit or the positions of planets in a solar system. All such systems can be simulated with the same technique. Most of these problems do not have discrete state values, like the high, low and undefined values used in the Flip-Flop simulation, but a real value, estimated by a floating point number.

Modelling processes with a non constant time step with a continuous simulation model results in a waste of computing power, since it is not necessary to recalculate the state continuously: stable parts need not to be simulated. This feature is provided by a *discrete time* simulator.

1.5 Discrete time simulation

To define a discrete time simulation, again streams are used to model the state of a particular wire in the circuit. In contrast with the continuous simulator, the streams now have explicit time stamps, to allow the functions to operate asynchronously on these streams. The processes may consume an element from one input list, without consuming an element from the other input lists. There is thus no direct relation between the position in the stream and the time stamp.

In Fig. 1.6 the Flip-Flop is specified with a discrete time model. The stream modelling a wire consists of data structures of the form `Until T V`. The meaning of this data structure (called a tuple for short) is that the wire will be in state V until time T; at time T the state changes instantaneously into the value described by the next tuple. By definition, the time stamps of the tuples in a stream are increasing: the time proceeds forward. The nand function operating on these streams generates an output tuple for the state up to the lowest time-stamp in its input streams. Since all other streams have a higher time stamp, the states of all these streams are defined by their first tuple.

By adding a constant to the time stamp of the output tuple, the delay of the nand-gate is modelled. In contrast with the continuous time simulator, the lists consist of the changes only: the length of the lists does not depend on the granularity of the time.

The representation of the Flip-Flop in Fig. 1.6 satisfies the requirement that the abstract program should remain unchanged while the abstract machine uses another data structure and corresponding `Nand` function.

```
> || ================ ABSTRACT MACHINE PROGRAM ================
> q, qb :: [state]
> q     = nand3 set   qb                        || The upper nand
> qb    = nand3 reset q                         || The lower nand
>
> || ============ INPUT OF ABSTRACT MACHINE PROGRAM ============
> set, reset :: [state]
> set   = [Until  7 L, Until 28 H ]             || A clock period
> reset = [Until 14 H, Until 21 L, Until 28 H ]
>
> || =================== ABSTRACT MACHINE =====================
> state  ::= Until timestamp threestate   || The state of a wire
>
> select2 :: [state] -> [state] ->
>            (timestamp,threestate,threestate,[state],[state])
> select2 xs ys = (yt,xv,yv,xs ,ys'), if yt < xt
>               = (xt,xv,yv,xs',ys ), if xt < yt
>               = (xt,xv,yv,xs',ys'), otherwise
>                 where
>                 (Until xt xv):xs' = xs
>                 (Until yt yv):ys' = ys
>
> delay :: timestamp -> timestamp;
> delay x       = x + 3
>
> nand3, nand3' :: [state] -> [state] -> [state]
> nand3' [] ys  = []                        || Terminate
> nand3' xs []  = []                        || Terminate
> nand3' xs ys  = Until (delay t) (nandfun xv yv) : nand3' xs' ys'
>                 where
>                 (t,xv,yv,xs',ys') = select2 xs ys
>
> nand3 xs ys   = Until (delay 0) X : nand3' xs ys
```

FIG. 1.6. The source code for a discrete time simulator.

The correctness of the discrete time simulator can be established by showing that the simulator does not deadlock, generates correct outputs, and makes progress in time. The simulator will not deadlock because of the following invariant: the consumption of a tuple from an input stream, will always result in a new tuple on the output stream. By using the theory of (Sijtsma 1989) again, the function `nand3` is +1-productive so the simulator will not deadlock. The simulator generates correct output values as long as all streams are ordered on their time stamp. It can be proved from the definition of `nand3'` that when the time stamps on the input lists are increasing, the output lists are ordered as well, hence all streams

are ordered. Because a non zero delay is added to the time stamp, the simulator makes progress in virtual time.

The invariant "each consumed tuple produces an output tuple" is also the weakness of the approach. Due to the loop in the definition of the Flip-Flop, tuples with increasing time stamp but identical state will start racing around the `Q` and $\overline{\texttt{Qb}}$, while the state of the circuit does not change. This is best observed on the calculated value of `q`:

```
> q = [Until  3 X, Until  6 H, Until  9 H, Until 10 H, Until 12 H,
>      Until 15 H, Until 16 H, Until 18 H, Until 20 H, Until 21 L,
>      Until 22 L, Until 24 L, Until 26 L, Until 27 L, Until 28 L,
>      Until 30 L, Until 31 L]
```

Although `q` is high from tick 3 until tick 20, there are seven tuples stating that `q` is still high at ticks 6, 9, 10, 12, 15, 16, and 18. The tuple at tick 12 is caused by the tuple at tick 6: `Until 6 H` in `q` causes a tuple `Until 9 L` on `qb`, which in turn causes the tuple `Until 12 H` on `q`. In the same way, the tuple at tick 9 causes the tuple at 15, and the tuple at tick 10 causes the tuple at 16.

The seemingly obvious solution of deleting tuples with an identical state is incorrect: it violates the invariant, and leads to an immediate deadlock of the program. Some of the redundant tuples can be avoided by *relaxing* the invariant: in the case of a nand, a low signal on one input fixes the output to a high value, regardless of the other input (it may even be undefined), on which the tuples may thus be ignored. When the nand is reprogrammed to ignore tuples on one channel as long as the other channel is low, far fewer tuples are generated. Still, the stable Flip-Flop has tuples running around, since both $\overline{\texttt{Set}}$ and $\overline{\texttt{Reset}}$ are high in the stable situation, requiring a redundant tuple to float through `Q` and $\overline{\texttt{Qb}}$.

The redundant tuples floating around are essentially caused by the distributed nature of the approach: the two nand circuits and wires operate completely autonomously which makes an empty tuple essential for the progress in the simulation (in (Chandy and Misra 1979) a similar effect is observed, which is solved by sending explicit NULL messages to keep a distributed simulator running). The problem can be solved by centralising the solution, giving rise to a simulator known as an event-driven simulator.

1.6 Event driven simulator

With respect to the previous simulator, two major changes are implemented: an explicit state of all wires is maintained, and there is a global list of things to happen in the future, the so called *events*. The events leads to new states, and possible new events. The Miranda source of the event driven simulator is listed in Fig. 1.7. The circuit description has been embedded in the definition of recalculate. This has altered the circuit description although the essential aspects have been retained.

```
> wire  ::= Set | Reset | Q | Qb
> state  == [(wire,threestate)]            || List of association pairs
> event ::= At timestamp wire threestate
> || ================= ABSTRACT MACHINE PROGRAM ================
> recalculate  :: wire -> (timestamp -> wire -> state -> event)
> recalculate  Q     = nand4 Set Qb
> recalculate  Qb    = nand4 Reset Q
>
> dependencies :: wire -> [wire]
> dependencies Set   = [Q]  || Wire Set used in def of wire Q
> dependencies Reset = [Qb] || etc. Can be derived automatically
> dependencies Q     = [Qb] || from definition of recalculate,
> dependencies Qb    = [Q]  || explicit for the sake of simplicity
> || ============ INPUT OF ABSTRACT MACHINE PROGRAM =============
> set           = [ At 0 Set L,   At  7 Set H,   At 28 Set L   ]
> reset         = [ At 0 Reset H, At 14 Reset L, At 21 Reset H ]
> initialevents = merge set reset
> initialstate  = [(wire,X) | wire <-[ Set, Reset, Q, Qb ] ]
> || =================== ABSTRACT MACHINE =====================
> select :: state -> wire -> threestate
> select st want     = hd [old | (have,old) <- st; want = have]
> update :: state -> wire -> threestate -> state
> update st want new = (want,new) : [(w,v) | (w,v) <- st; want ~= w]
>
> sim :: [event] -> state -> [event]
> sim []     st = []                                  || Termination
> sim (e:es) st = sim es st, if select st wire = what ||  IGNORE
>               = e : sim es' st', otherwise         || Process event
>                 where
>                 (At time wire what) = e
>                 es'  = merge es (sort more)        || New events
>                 st'  = update st wire what         || New state
>                 more = [ mkevent out | out <- dependencies wire ]
>                 mkevent wire = (recalculate wire) time wire st'
> delay :: timestamp -> timestamp
> delay t = t+3
>
> nand4 :: wire -> wire -> timestamp -> wire -> state -> event
> nand4 x y t w st = At (delay t) w (nandfun (select st x) (select st y))
>
> main  = sim initialevents initialstate
```

FIG. 1.7. The source code for an event driven simulator.

An event is represented by a three tuple `At time which what`. The tuple tells at what time, which wire will get what value. The list of events is sorted on increasing time stamp, so the event with the lowest time is

handled first. The time of the lowest event is also the current time of the simulation, and since the time should increase, newly generated events should have a time stamp larger than or equal to the current time, also known as the causality condition: an event can have consequences for the future, but no consequences for the past. The function `sim` traverses the list of events recursively, each time producing a new state based on the old state and the consumed event.

An event may change the state of the circuit, and when the state of one of the wires is changed, all components connected to that state are required to recalculate their output value, generating new events for their output wires. The new events are merged into the old event list before calling the simulator to consume the rest of the events. Note that in this example program the circuit is specified twice. In the definition of `recalculate` and in the definition of `dependencies`. This last function tells which wires have to be recalculated on a change of a state on a specific wire. This last function is in fact the inverse of the former, and can be derived automatically, but both are defined for the sake of simplicity.

An event that does not cause any change in the state may be ignored, as is done on the line marked *IGNORE*. An optimisation which is not allowed in the discrete time simulator of the previous section, because it would cause a deadlock of that algorithm. The event driven simulator does not deadlock because all events are managed centrally: there are still future events to continue the calculation. Without this optimisation, the event driven simulator would have the same poor performance as the previous discrete time simulator.

The output of the event driven simulator contains the state changes on all the wires as a sorted list:

```
> main = [At 0 Set L, At 0 Reset H, At 3 Q H, At 6 Qb L, At 7 Set H,
>         At 14 Reset L, At 17 Qb H, At 20 Q L, At 21 Reset H,
>         At 28 Set L, At 31 Q H, At 34 Qb L]
```

1.7 Discussion: summarising the simulation algorithms

The four algorithms presented in the previous sections describe the basic principles of simulation. The second algorithm is the one used in all continuous time simulations, because of the fixed time step. The discrete time and event driven algorithm are used for problems with varying time steps. However it is possible to simulate a continuous time problem with a discrete time simulator, or vice versa, at the cost of decreasing performance. Discrete simulations are sometimes parallelised by using the discrete time algorithm, it has a distributed nature, but other algorithms can be parallelised as well (Misra 1986, Overeinder *et al.* 1991). There are some important differences in terms of efficiency, expressiveness and potential parallelism. A comparison on base of these aspects is presented below.

1.7.1 *The efficiency*

The demand driven algorithm is inefficient (both in space and in time). In the worst case, the algorithm needs execution time that is exponential in the in the number of components and the length of the virtual time (construct a Flip-Flop with *three*-input nand gates and connect the second and the third input of the upper nand to $\overline{\texttt{Qb}}$, and the first and the second input of the lower nand to `Q`: the values of `q` and `qb` will be calculated twice, recursively, leading to an exponential time behaviour).

The time requirements of the continuous time algorithm are linear in the length of the simulation run, and inversely proportional to the time step δ_t. A discrete simulation can be performed using a continuous simulator (δ_t should be set to the greatest common denominator of all delays in the circuit), but it is inefficient, since there are many unnecessary recomputations.

Although the discrete time algorithm deals better with problems with a non constant time step, inactive parts of the model need still to be recomputed to prevent the simulator from deadlocking. The worst case time behaviour is for this reason identical to that of the continuous time algorithm.

The event driven algorithm deals with *all* inefficiencies, the execution time is linear in the number of executed events, the number of changes of the state in the circuit. For this reason the event driven algorithm is widely used for discrete time simulations.

1.7.2 *The expressiveness*

All algorithms can be used to simulate all types of problems, although simulations of a discrete time problem with a continuous time simulator or vice versa are rather inefficient. There is another difference between the algorithms that is related to the correctness of the simulation algorithm. In the example of the Flip-Flop a constant delay is introduced by the nand gate: the nand gate delays the output signal with 3 clock ticks. However, in realistic circuits it is common that the delay is not fixed, but that the delay depends on the state of the circuit. The example of Fig. 1.8 shows a more realistic scheme of how a simple buffer behaves. In this example, the buffer needs 5 time steps to drive a signal high, and only 1 time step to bring the signal back to low again. A pulse entering the buffer, will result in a shorter pulse on the output of the buffer.

The first algorithm computes in a demand driven fashion. Consequently, the delay has to be known before the state is calculated. This implies that the delay can only depend on the time in that part of the program, and not on the state. The buffer of Fig. 1.8 can thus not be modelled using this algorithm.

The continuous time algorithm introduces the delay as a fixed number of `X`'s at the start of the output list. It is possible to make the delay

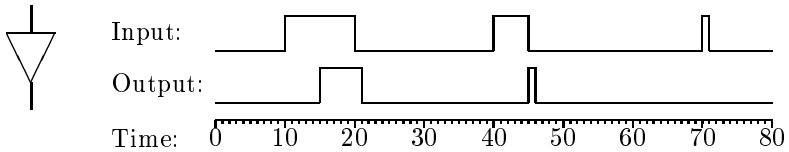


FIG. 1.8. A simple buffer circuit (`Output := Input`), that needs more time to get a signal high, than to bring it to low again. Consequently, pulses are shortened, while a short pulse disappears.

dependent on the state of the wire, by checking if the wire contains a swap from `L` to `H` (or vice versa), and to generate a variable number of output elements as response (extra output elements for more delay, fewer output elements for short delay). Still, it is possible to prove that the simulator does not deadlock in that case (see (Muller 1993) for more details).

In the discrete time algorithm, the function `delay` adds the constant 3 to the time. This function may be changed to an arbitrary function as long as `delay` satisfies two properties that follow directly from the correctness proof of the algorithm: progress in virtual time should be guaranteed (`delay` $t > t$) and the streams should be ordered ($t_x > t_y \Rightarrow$ `delay` $t_x >$ `delay` t_y). The first condition forbids the simulator to run backwards (in fact the causality condition), and the second property guarantees that the lists remains ordered: an unordered list of output tuples has no meaning. A `delay` that directly depends on both state and the time is incorrect: suppose that the delay function returns $(t+5)$ on a high input, and $(t+1)$ on a low input, in that case the second property is not satisfied (since `delay H` $10 \not> $ `delay L` 12).

In the event driven algorithm, the only constraint is that an event generated at time t should have a time stamp greater or equal than t (the causality condition). Consequently, two consecutive events I_1, I_2 on an input, with `Time`$(I_1) <$ `Time`(I_2), can cause two output events O_1, O_2 with `Time`$(O_1) >$ `Time`(O_2), implying that the second event has overtaken the first one. This is sometimes the intended result, but most of the time, the result is disastrous. Although the event list provides the most flexible solution in terms of the delay, it is also the most dangerous implementation: it offers a designer the possibility to shoot in his own foot. It does not only support assignments, it even contains assignments *somewhere in the future*. Functional programmers advocate that it is hard to reason about assignments, it is illustrated here that it is even harder to reason about future assignments. An event list should be used carefully so that common causality rules are not violated.

1.7.3 *The potential parallelism*

The first simulation algorithm has an exponential complexity, the parallelisation is thus trivial (ordinary divide and conquer suffices). The continuous and discrete time simulators are trivially parallelised, as well. Both lead to process networks (Kahn 1974) that can be mapped onto a multi processor machine. The communications and computations are decentralised, so as long as the communication graph of the process network can be mapped on the physical machine, both the continuous and discrete time simulators are perfectly scalable. Depending on the number of start elements on the various lists, the communication and computations can even be pipelined. The event driven simulator is not parallel at all. The central event list needs to be stored somewhere, and even events with an identical virtual time cannot be executed in parallel in this implementation (because the order in which events are applied might result in different values). For this reason discrete simulations are always parallelised using the discrete time algorithm. Heuristics are then applied to reduce the number of redundant synchronisations (see for example (Vries 1990)), but in worst case the algorithm is as inefficient as a continuous time simulation.

1.8 Conclusion

The abstract machine models of four simulation algorithms have been specified. These are for a demand driven, a continuous time, a discrete time and a event driven simulation. Three of the models have a distributed control that allows for a parallel implementation of the simulation problem, while the fourth model has a centralised nature. To extend the insights on the abstract machine models, they have been specified formally as functional programs. These formal definitions allow us to reason about the performance (parallelism, efficiency) and correctness (deadlock freedom, simulation results).

Demand driven simulation is inefficient, and is consequently directly parallelisable. Continuous time simulations are trivially parallelised (in fact many applications that are currently executing on todays parallel machines are continuous simulation problems). Parallelising discrete time simulators is much harder, the version that is easily distributed is inefficient (in the worst case as inefficient as the use of a continuous time scale simulator), the one that is efficient cannot be distributed because of the global control. Parallelisation of discrete time problems in general is hard, but for certain classes of simulation problems (which can be formulated using the discrete time algorithm without loosing too much efficiency) an efficient parallel simulation is feasible.

Acknowledgements

We thank Wim Vree for his comments on a draft version of the paper.

REFERENCES

R. S. Bird and P. L. Wadler, (1988). *Introduction to functional programming*. Prentice Hall, New York.

K. M. Chandy and J. Misra, (1979). Distributed simulation: A case study in design and verification of distributed programs. *IEEE transactions on software engineering*, SE-5(5):440–452.

R. J. M. Hughes, (1985). Lazy memo-functions. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 129–146, Nancy, France, Sep. Springer-Verlag.

G. Kahn, (1974). The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. North Holland.

P. H. J. Kelly, (1989). *Functional programming for loosely-coupled multiprocessors*. Pitman publishing, London, England.

J. Misra, (1986). Distributed discrete-event simulation. *Computing surveys*, 18(1):39–65.

H. L. Muller, (1993). *Simulating computer architectures*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, February 26.

B. Overeinder, L. O. Hertzberger, and P. M. A. Sloot, (1991). Parallel discrete-event simulation. In W.-J. Withagen, editor, *3rd Computer systems*, pages 19–30, Eindhoven, The Netherlands, May. Univ. of Eindhoven.

B. A. Sijtsma, (1989). On the productivity of recursive list definitions. *ACM transactions on programming languages and systems*, 11(4):633–649.

D. A. Turner, (1985). Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep. Springer-Verlag.

D. A. Turner, (1990). *Miranda system manual*. Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England.

W. G. Vree, (1989). *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec.

R. C. de Vries, (1990). Reducing null messages in Misra's distributed discrete event simulation method. *IEEE transactions on software engineering*, SE-16(1):82–91.